**CERIAS Tech Report 2018-01**
**Automated Differential Testing for Energy-Efficient Control Software**
by Hongjun Choi
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

# Automated Differential Testing for Energy-Efficient Control Software

Hongjun Choi
choi293@purdue.edu
Purdue University
United States

Bruce V Nguyen
nguye167@purdue.edu
Purdue University
United States

Sayali Kate
skate@purdue.edu
Purdue University
United States

Xiangyu Zhang
xyzhang@cs.purdue.edu
Purdue University
United States

Dongyan Xu
dxu@cs.purdue.edu
Purdue University
United States

## ABSTRACT

Cyber-physical systems (CPS) are integrated systems of computer-based algorithms and physical components interacting with environmental effects. In such systems, autonomous behaviors and overall performance mainly depend on a control software. Thus, it is crucial to test and analyze the control software of the CPS in various perspectives. One of the critical perspectives is energy efficiency because many cyber-physical systems (e.g. unmanned aerial vehicles, autonomous cars, health-care devices) operate with limited energy sources such as batteries. In this paper, we propose CPSDiff: an energy-aware differential testing framework that generates test inputs to expose the maximal difference between two control programs in energy consumption. Our test generation technique uses meta-heuristic searching to find the input that maximizes the energy consumption difference. The difference-revealing ability of our technique outperforms the random search algorithm and hill-climbing search algorithm. Our evaluation on two popular unmanned aerial vehicle control programs provides a detailed comparison of their energy consumption under the same condition with a universal robotics simulator; CPSDiff found the input which exposes maximum battery consumption difference of around 47%.

## 1 INTRODUCTION

A cyber-physical system (CPS) is a combination of physical and cyber subsystems. For example, robotic vehicles, such as self-driving cars [5, 10, 12] and unmanned aerial vehicles (UAVs) [3, 11], include a control program, various sensors, and actuators from the two different domains. The hybrid (discrete and continuous) components closely interact with each other and perform mission-critical tasks or autonomous operations. Among those several components in CPS, the control program plays a critical role. Autonomous operations and complex missions require such a control program to take high-level commands, process sensor data, and control actuators. Therefore, the efficiency of a CPS significantly relies on the performance of the control program.

The efficiency of a CPS can be measured in various perspectives. One of the important metrics is *energy consumption* because many CPS operate with limited energy sources. It is critical for a battery-powered system to reduce power consumption in order to achieve a longer operation time. *Different control mechanisms may show distinct energy consumption for the same mission.* An inefficient control mechanism may consume significantly more energy than a well-optimized one, and it may even lead to vehicle damages/failures because of energy deprivation and deficiency, especially in long-time continuous operations. [1, 2]

Despite the importance, there has not been a practical solution for energy efficiency testing for CPS. Existing approaches in general energy efficiency testing [23, 26, 34] are not applicable to CPS since their usage scenarios are primarily in the cyber domain only, such as an extensive file I/O and cryptographic operations, or focus only on specific and pre-defined use cases. For energy-aware CPS testing, the testing mechanism should consider interactions between the cyber and physical domains and various environmental disturbances. In addition, because of the nature of trade-offs between control properties [35], it is difficult to design general test oracle of energy consumption to support various scenarios. Otherwise, the engineers are required to identify important usage scenarios and develop test oracles [16] since these tasks require a substantial amount of manual effort and deep domain knowledge.

There have been many studies in program analysis and testing for conventional systems, such as cloud [24], web server [18], and mobile systems [38]. However, CPS are fundamentally different from such systems. A CPS runs on a micro-controller and orchestrates sensors and other physical components with external disturbances in real-time with non-determinism. The behavior of the control program is hardly analyzed and tested without a well-configured infrastructure or a real physical operation environment. Furthermore, conventional control-domain tools, such as MATLAB Simulink [39], are mostly design-time testing tools and they require a specific model although there are many cases where a control model is unavailable or correctly acquiring it from the control program is infeasible in practice (e.g., legacy programs and proprietary software). Formal validation and verification approach [22, 34] would be one way to analyze CPS, but they are known to work only on small-scale software and suffer from the state explosion problem [22].

In this paper, we overcome the above challenges and propose a new practical approach to testing and comparing control software in the perspective of energy consumption. We were motivated by the fact that different control programs which have different control mechanisms or different versions of the programs cause behavioral differences of the target system. In addition, we observed that under certain types of operation or missions of a CPS, such differences

become more significant. Subsequently, the CPS consumes varying amounts of energy proportional to the differences.

Based on the above observation, we propose a framework, called CPSDiff, that enables *differential testing for control programs to automatically reveal performance differences in CPS, especially regarding energy consumption.* Given two control programs, CPSDiff simultaneously executes two programs in a black-box approach under the same physical simulation environment. Specifically, we mechanically generate random test inputs and apply those into the two programs and then compare the results based on a number of properties. Since our methodology is automatic black-box testing [17], CPSDiff does not require accessing the internals of the control programs. CPSDiff executes the programs multiple times with automatically generated inputs on top of a universal robotics simulator. The simulation supports various types of physical devices and environment effects, as such the generated inputs resemble real physical inputs.

CPSDiff searches for the *best input* which exposes the maximum in energy difference. The large difference, the more effectively the input can reveal the difference of control software quality. To find the maximum difference, CPSDiff efficiently explores the input space by using our *adaptive random searching algorithm* and it can successfully find the differences efficiently, compared to other searching algorithms such as a random and a hill-climbing algorithm.

We applied CPSDiff to a real-world CPS system, which is a commodity UAV with two popular control programs (ArduCopter [4] and PX4 [9]). Our evaluation results show that CPSDiff is able to successfully generate test cases which expose significant differences in energy consumption between the two UAV control programs.

The contributions of this paper are summarized as follows:

- We propose a differential testing framework for CPS control programs to expose their differences in energy consumption.
- To our best knowledge, we propose first energy-aware differential testing in the CPS domain. In the perspective of energy consumption, we compare control programs with automated test inputs and search the case which reveals potentially maximum energy difference of the systems.
- We set up a universal robotics simulation framework, Robotic Operating System (ROS) and Gazebo simulation, which provide comprehensive simulation environments to remove physical cost and support systematic testing for different control software under the same simulation condition.
- We develop new meta-heuristic search algorithm which utilizes a novel test strategy to expose a maximal difference. We apply CPSDiff to a practical use case, two different UAV control programs, ArduCopter and PX4, and provide quantitative evaluation results. The detailed results are available in §6.

The remainder of this paper is organized as follows. §2 provides the background and our motivating example of UAV systems. §3 gives an overview of CPSDiff. §4 describes the details of our approach, and §5 presents practical challenges we solve. §6 presents the implementation and evaluation of our research. Finally, §7-8 outlines related work and concludes our paper.

# 2 BACKGROUND AND MOTIVATION

In this section, we provide a background of CPS with UAV as an example and our preliminary experiments to motivate CPSDiff.

## 2.1 UAV system overview

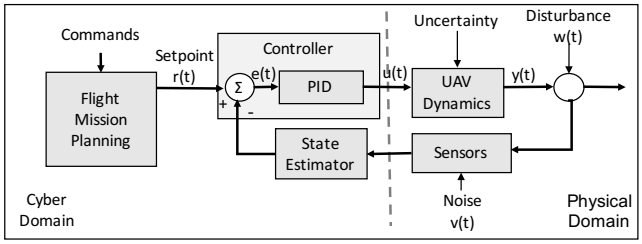



**Figure 1: A typical UAV control system architecture.**

As shown in Figure 1, a typical UAV system is composed of several components. The flight mission planning component takes high-level commands from a user and provides *setpoint*, a desired value such as a target altitude, to the controller. Taking the setpoint as a reference value, The goal of the controller is to maintain a stable flight with accurate tracking of the desired trajectory. For that, it attempts to reduce the error between the reference value and the measurement from a sensor. For example, a typical control mechanism, called proportional–integral–derivative (PID) controller [35], continuously calculates the error $e(t)$ between a desired setpoint $r(t)$ and a measured value. The proportional, integral and derivative terms attempt to eliminate the error over time by adjusting the control input $u(t)$ in the feedback loop. In a typical UAV system, the control inputs are motor pulse width modulation (PWM) signals that are used to adjust the rotation of motors and to control the speed and the attitude of the vehicle. Measurements from various sensors are then transferred to the controller as a feedback in the control loop and the feedback is used as a next input.

UAV is a hybrid system that exhibits both a continuous (in the physical domain) and a discrete behavior (in the cyber domain). In the physical domain, there are always external disturbances, such as wind and noises from different sensors. A *controller* and an *estimator* handle the disturbances to provide better responsiveness. In the cyber domain, UAV software supports various flight modes (e.g. auto, manual, and stabilized modes) for different flight missions. The *flight mission planning* component implements discrete behaviors of the UAV and an intelligent algorithm to handle high-level actions to improve the overall performance of the system.

A control software is fairly complex and has many different control properties. For example, *stability* is an important property of a control system. In a high-level description, a system can be considered as stable if it remains in a constant state unless affected by an external impact and returns to the constant state when the external impact is removed. Another important property is *responsiveness*. It is about how fast the system approaches a target state. In order to be responsive, a controller may introduce a large overshoot, and thus the system may become unstable and consume more energy. However, since it also reduces a settling time to get to the target point, energy consumption may be decreased because of the

saved time. Consequently, there exists no optimal control algorithm which shows the best performance to satisfy every property. This motivated us to come up with the new idea to automatically test control systems through a novel differential testing since comparing and understanding trade-offs of control software is important for choosing the better control software for diverse mission scenarios.

## 2.2 Differential Testing

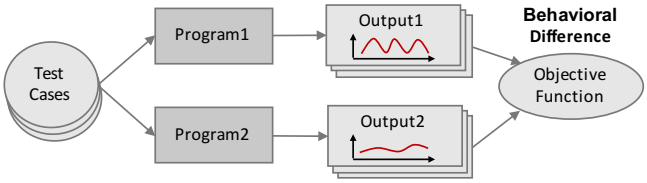



**Figure 2: Differential Testing. Two programs run with the same inputs and two outputs under the same condition are compared to show differences.**

In general, differential testing [29] compares the execution of two programs side by side using the same inputs, as shown in Figure 2. Our testing mechanism automatically generates the inputs, run the two programs with the inputs, and evaluate the quality of the programs' outputs for a comparison.

As discussed previously, because of complex and conflicting properties of the control system, generating test cases and evaluating test results for the control system require a huge effort. Without deep domain knowledge, defining a test oracle is a challenging task. Our test technique does not require the domain knowledge. We compare two control programs for showing the differences between their behavior, without the test oracles defined by a domain expert. The programs under test could be of two different versions or could have a totally different design and implementation to support the same function. While the differential test can be applicable to the comparison of two unknown qualities of programs, if the quality of one program is well-evaluated already, it can be used as *base program* to compare others.

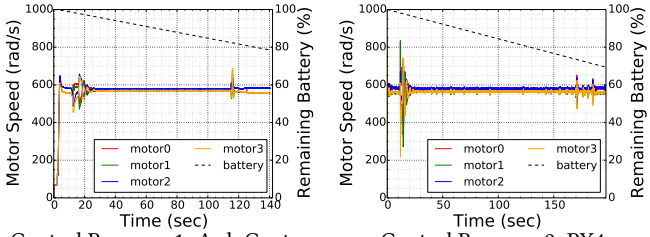
## 2.3 Motivative Example

(a)Horizontal Flight (b)Zigzag Flight

**Figure 3: Flight Missions. The UAV flies following the pre-defined trajectories (white line) and moves to the target waypoints (green numbers).**

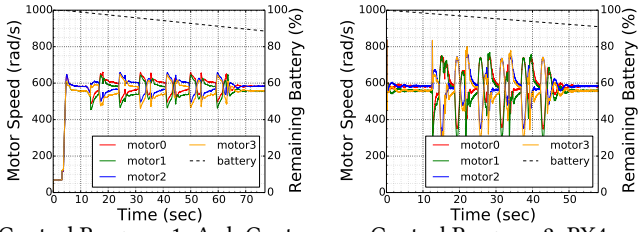
In order to expose the behavioral differences of control software, we performed preliminary experiments with two different proprietary quad-rotor control programs: ArduCopter and PX4. We used the same hardware model, 3DR IRIS+ [6], for the two control programs. The control software communicates with a ground control system (GCS) via Micro Air Vehicle Communication (MAVLink) protocol [8], and the GCS is able to send flight control messages and missions to the vehicle either at runtime or offline. In our experiments, we manually planned several flight missions and applied them to the two control programs. During the flights, we collect flight information for the motor speed and the energy consumption.

Figure 3 shows two different flight missions, where the numbers represent a mission command sequence and the white line represents a planned trajectory the UAV will fly on. The left mission is *horizontal flight*. The UAV will take off from the home position (number 0) to an altitude of 20m (number 1) and then horizontally fly 1km to the endpoint (number 2). The second mission is *zigzag flight*. The UAV will take off from the home position and then move left and right in a zigzag fashion.




Control Program 1: ArduCopter Control Program 2: PX4

**Figure 4: Comparison of motor speed and battery consumption in the *horizontal flight* mission**




Control Program 1: ArduCopter Control Program 2: PX4

**Figure 5: Comparison of motor speed and battery consumption in the *zigzag flight* mission.**

Interestingly, there were significant differences in the battery consumption of the two control programs during several missions. As shown in Figure 4, ArduCopter finished the *horizontal flight* mission early and consumed less energy than PX4. Upon the mission completion, ArduCopter consumed around 21% of the total capacity of the battery, while the PX4 consumed 31%. The difference in the amount of consumed battery is roughly 10%. However, in the *zigzag flight* mission in Figure 5, PX4 outperforms ArduCopter. At the end of the mission, PX4 consumed 8%, but ArduCopter consumed 11%.

Based on these preliminary experiments, we observed that the control programs show different performance depending on the type of the mission.

**Limitations of the Existing Testing Tools.** Model-based testing with Simulink [39] relies on models to generate test scenarios and oracles in order to find interesting test inputs. A domain expert should carefully choose the scenarios and manually test the system. However, as described in §2.1, the control program is complex, and thus obtaining an accurate control model from the control program is practically infeasible. Moreover, manually testing such complex programs requires a substantial amount of effort. For example, ArduCopter and PX4 have around 600 and 400 control parameters, respectively. Different parameter settings affect the control programs to have diverse internal behaviors and energy consumption. Therefore, experts should subsequently choose test cases and oracles to test the system on every different configuration manually.

**Our Approach.** Our objective is to provide an automated testing tool, CPSDiff, which requires no accurate control models and test oracles. CPSDiff uses a differential testing technique: it systematically executes two control programs and exposes their maximum performance differences. The tool also requires no deep domain knowledge to design test scenarios. We use a high fidelity robotics simulator to test the CPS control software with automatically generated missions under the potentially hostile physical environments.

## 3 OVERVIEW

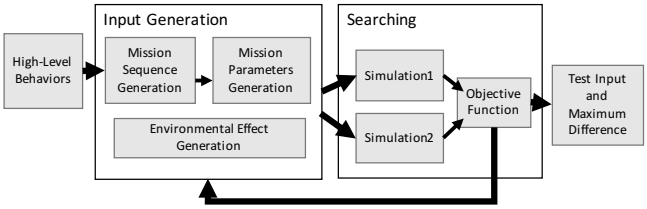



**Figure 6: An overview of automated differential testing of CPSDiff**

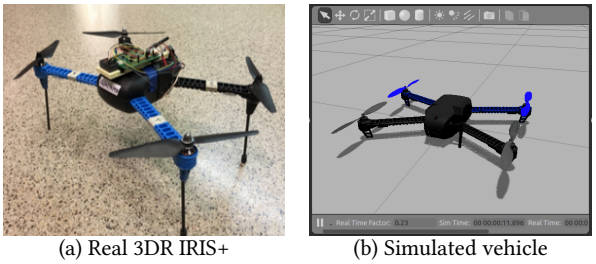
The overall workflow of CPSDiff is presented in Figure 6. CPSDiff requires two different control programs for the testing. The control programs differ in that they may implement different control mechanisms, have different internal components, and support different sets of autonomous actions, etc. For a fair comparison, we configure the control programs to operate in the same *flight mode*. The flight mode defines how the system operates, including manual, autopilot, or mixed operations. For example, in ArduCopter and PX4, there are many built-in flight modes; ArduCopter has 14 and PX4 has 5 modes. In our experiments, we use the *autopilot* mode for both of the control programs, which controls the UAV based on a pre-programmed mission script transmitted from GCS.

In the second step, CPSDiff generates an input mission and environment effects. An input mission is defined by two property groups. The first is a sequence of *mission commands*, which includes various types of flight behaviors, such as takeoff, waypoint flight, landing and loitering. The commands potentially have causal relationships with each other that can be represented by a state transition diagram. CPSDiff uses these state transitions for the automatic generation of appropriate test cases which have different sequences of commands. The second is mission command parameters. Once the sequence of commands is chosen, CPSDiff generates parameter values for each mission command. For example, the waypoint command has three parameters: latitude, longitude, and altitude. The input generator populates the parameters with values based on the given input space. Besides the input mission, CPSDiff also generates environment effects if required.

In the searching step, CPSDiff runs the two simulations simultaneously under the same condition except control programs, and then profiles the flight information while performing the input mission. Our objective function evaluates the quality of the output after the mission. The evaluation data allow our searching mechanism to find the maximum difference between the two control programs in energy consumption. Our meta-heuristic search algorithm run multiple iterations for a given time budget (i.e. number of search iterations).

## 4 DESIGN

In this section, we discuss the universal robotics simulator and the further details of each component of CPSDiff.



(a) Real 3DR IRIS+ (b) Simulated vehicle

**Figure 7: A UAV used for control software testing.**

### 4.1 Universal Robotics Simulator

A highly reliable test environment is essential in CPS testing. Since the control software always interacts with physical devices and external environmental effects, field testing with a real hardware platform would be the most accurate way for testing. However, field testing requires expensive hardware and a significant amount of time to test various scenarios. In addition, unexpected behaviors or failures during the tests can damage devices physically and may cause critical safety issues. Thus, instead of a real hardware platform, we use a high-fidelity simulation framework, Gazebo [7].

Gazebo is the most popular 3D robotics simulator in the robotics research, and has a modular design to support different kinds of robotics and their components. It describes physical properties of a robot using the Universal Robotics Description Format (URDF) or the Simulation Description Format (SDF). A User can develop his/her own physical vehicles or objects with this standard XML format. This way, Gazebo is able to simulate virtually any type of physical objects as long as we define our objects' physical properties and implement custom components as plugins. In our case, we use 3DR IRIS+ quadcopter shown in Figure 7 for the target hardware

model and the corresponding URDF. All the components such as sensors, UAV dynamics, and external influences are simulated by Gazebo plugins or physics engines in the Gazebo simulator.

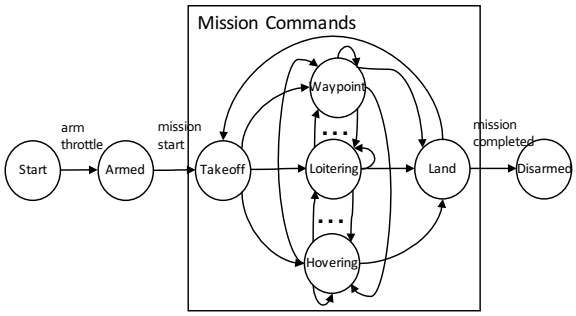



**Figure 8: Mission state transition diagram. The mission commands have causal relationships and a mission input includes an appropriate sequence of mission commands.**

## 4.2 Test Input Generation

We use a mission and environment effects as a test input. Generating the input mission consists of two steps: generating a sequence of mission commands and populating parameters to each generated mission command. We denote a test input vector as $I = \{t, w\}$ where $t$ is the sequence of missions commands with parameter values and $w$ represents the wind effects.

**Mission Sequence Generation.** In our test scenarios, the UAV would undergo a series of pre-programmed behaviors. For example, the UAV starts the mission with vertical takeoff ($T$) from the same home position in each scenario and performs a sequence of mission commands such as waypoint ($WP$), loitering ($LO$), hovering ($HO$). The mission completes when the UAV safely lands to the target position with the land ($L$) mission command. Figure 8 shows the possible transitions between the mission commands. CPSDiff generates a sequence of an arbitrary number of input mission commands in a random manner. Using the mission state machine, it traverses the topology randomly, and each traversed path represents a sequence of input mission commands as shown in Table 1.

**Table 1: Randomly generated a sequence of mission commands.**

| Test No. | A sequence of mission commands |
|---|---|
| T1 | $T \rightarrow WP \rightarrow LO \rightarrow WP \rightarrow L \rightarrow T \rightarrow LO \rightarrow L$ |
| T2 | $T \rightarrow HO \rightarrow WP \rightarrow WP \rightarrow L$ |
| T3 | $T \rightarrow LO \rightarrow L \rightarrow T \rightarrow L \rightarrow T \rightarrow WP \rightarrow L$ |

**Mission Parameter Value Generation.** Every mission command has its own set of parameters. MAVLink supports at most 7 parameters for each command. For example, $WP$ command requires latitude, longitude and altitude as the parameters. We test the behaviors of UAVs within a *virtual fence*, and thus each parameter is assigned a value range: $[min_{lat}, max_{lat}]$, $[min_{lon}, max_{lon}]$, $[min_{alt}, max_{alt}]$. For each parameter, CPSDiff selects a value from a uniform distribution within the given value interval.

In order to test with a realistic environment, CPSDiff also inserts wind and wind gust effects $w$. The environmental effects are simulated in the Gazebo plugins and described with the plugin parameters in URDF. The *wind* plugin has the following parameters: direction, force. *Windgust* has direction, duration, force and start time.

## 4.3 Objective Function

CPSDiff runs two simulations with the generated test input vector $I$ in parallel and profiles the information of vehicles' behaviors during the simulation, e.g., flight time, distance, motor speeds and battery consumption over time. When the mission is completed in both the control systems, the output values (i.e., battery consumption) are compared. Our test objective is to find the input that maximizes the performance difference. In this paper, we compare the amount of consumed energy for the performance difference.

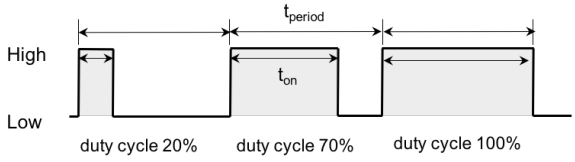



**Figure 9: Examples of PWM signals.**

**Energy Consumption.** Many battery related works [20, 31, 32] focus on estimating the energy consumption of battery-powered devices, using accurate battery models or additional hardware devices to measure real energy consumption. However, we use a simple measurement method, as our purpose is just to compare the two results rather than accurately measure the consumption. Intuitively, the control program regulates control signals and in the UAV system, the signals control the speed of multiple electric motors to generate a torque. In other words, the electric motor control is a main role of the control software. Thus, we simplify the battery model by taking an advantage of this fact: the energy consumption is proportional to motor speeds and the control programs only generate pulse-width modulation (PWM) signals which control the motor speeds directly. This way, CPSDiff estimates the energy consumption by only relying on the motor speeds.

Speeds of the UAV DC Motors are controlled by the periodic PWM signals. According to the different duty-cycle $c = t_{period}/t_{on}$ of PWM as shown in Figure 9, the switch turns on and off and the effective voltage $V_{eff}$ is applied to the motor. Our target system, 3DR IRIS+, has a 5100 mAh polymer (LiPo) battery pack and we assume average 50A current is drawn at 100% duty cycle, i.e., full throttle. During the simulation, CPSDiff collects the information on energy consumption by sampling the current. The total remaining battery, $B_{rem}$, is then calculated as given below:

$$B_{rem} = B_{tot} - \sum_{k=1}^{n} I_k (t_{k+1} - t_k)$$

The current $I_k$ is sampled at every time $t_k$ and the consumed current in each sampling period from $t_{k+1}$ to $t_k$ is derived. The sum of consumed current is subtracted from total battery capacity $B_{tot}$ to

get the remaining battery $B_{rem}$. After the simulation is completed with mission completion, CPSDiff compares the remaining battery to assess the quality of output. Our search algorithm uses this output to find best results. We do not limit our objective function to the difference in energy consumption. Depending on the application, CPSDiff can be set up to use the different objective functions to show differences in other properties such as stability, overshoot, responsiveness, etc.

## 4.4 Search Algorithm

Our goal is to find mission inputs by maximizing the objective function value, the difference of energy consumption. Since our input space is too large, brute-force approach is infeasible. Therefore, CPSDiff uses a meta-heuristic search algorithm to efficiently find the best input.

To solve this problem, intuitively, the first simplest approach is random search with bounded trials. Because of the notion of randomness, the pure random search is extremely explorative, which means that the algorithm mostly explores globally to find the optimal value. Another opponent approach is Hill-Climbing search algorithm, largely doing local improvements in the limited region. However, if the limited region is not promising, the local optimum is not a good solution in the global space.

In the CPS and control programs, the test input space is continuous and hybrid. For example, the input of a UAV system consists of two level: discrete mission commands and its continuous parameters. Especially, every value for the continuous parameters is impossible to be visited, and the best possible output can not be guaranteed unless the entire space is visited.

In this context, we propose a practical search algorithm to balance between exploration and exploitation. Our algorithm uses adaptive random search [21] with novel testing strategies to address the above limitations. The CPS simulation usually takes a long time because of some unique properties such as real-time simulation and heavy computations. Therefore, the key strategy in the given limited time and resource is to improve *input quality* instead of running lots of simulation. If we are able to select high-quality inputs which are more likely to generate better output, the output will approach to the optimal value more quickly.

The algorithm 1 represents a adaptive random search. Initially, the algorithm takes an initial test input and mission-related information (i.e. a sequence of mission commands and input range vectors) as an input, which was generated during the test input generation step. Line 11-19 correspond to the main loop to find the maximum difference. At line 12-13, two simulations generate energy consumptions with the test input $I$ and then $I$ is stored in the history $H$. At line 14-15, when the difference is higher than the current best solution, the new solution is selected. Otherwise, new test input $I$ is generated by using adaptive random selection algorithm at line 16.

**Test Strategies.** Basically, the algorithm utilizes the random search. However, we use additional test strategies to speed up the search procedure and distribute selected inputs evenly across the input space. While the original random selection algorithm selects a single input randomly, our algorithm selects an input among k randomly selected candidates based on the quality of the input.

**Algorithm 1** Adaptive Random Search.

```
1:  N ← number of iterations
2:  M ← sequence of mission commands
3:  R ← input range vectors of parameters of mission commands
4:  S1, S2 ← simulation with program1 and program2
5:  t ← GenerateInitialRandomInput(M, R)
6:  w ← GenerateRandomWindEffect()
7:  I ← {t, w}
8:  H ← null
9:  BEST ← I
10:
11: while N >= 0 do
12:     o1, o2 ← outputs generated by S1 and S2 with test input I
13:     H ← I
14:     if Diff(o1, o2) > BEST then
15:         BEST ← I
16:     t ← AdaptiveRandomSelection(M, R, H)
17:     w ← GenerateRandomWindEffect()
18:     N ← N − 1
19: return BEST
```

**Algorithm 2** Adaptive Random Selection.

```
1:  M ← sequence of mission commands
2:  R ← input range vectors of parameters of mission commands
3:  H ← history of t
4:  k ← number of randomly generated candidates
5:
6:  randomly generate k candidates c_1, c_2, ... c_k
7:  qscore = 0
8:  for each candidate c_i do
9:      s_i ← calculate qscore for c_i
10:     if s_i > qscore then
11:         t = c_i
12: return t
```

- **Strategy1: Evenly Spread.** Intuitively, adjacent test cases are more likely to result in similar results. In other words, given a previously executed test case, new test cases located away from the one is more likely to expose better results. CPSDiff takes an advantage of this intuition to generate the test cases which are more evenly spread across the input space in order to test different area. Among the $k$ candidates, for each candidate $c_i$, distance $d_i$ is determined from the closest previous test case and the candidate with the largest distance $d_i$ has higher input quality.
- **Strategy2: Sudden Behavioral Change.** In autonomous missions of CPS, the control software has to control the physical devices (i.e. motor speed in UAV case) according to the expected behavior. When the mission requires a more sudden change in its behavior, the program may require a better control ability than monotonous and stable movement. When we choose an input, we score the input based on this property. For example, UAV system flies to a certain target position. The target position is along the current flight direction, the control program does not require to adjust yaw (head direction). On the contrary, if the UAV flies to the opposite direction or performs a sudden rise, the control program needs to do additional operations. In this context, we evaluate the *input quality*. One example we implement is

turning direction. The sharper turnings are more likely to cause performance difference. When we select next target position, among the random k candidates, for each candidate $c_i$, turning angle $\theta_i$ is calculated with two previous mission waypoints on the planned trajectory. The candidate with the smaller $\theta_i$ has higher *input quality*.

**Input Quality.** We measure the *input quality* which is more likely to generate better output by using the above strategies. The algorithm 2 shows the adaptive random selection with the score of input quality. Specifically, the algorithm selects $k$ candidates randomly (line 6). Then, the highest scored input is selected based on its quality score, *qscore* (line 8-12). The *qscore* is calculated using the above strategies.

When we use more than one strategy and metric to measure the quality of an input, we can use weighted score sum. $w_j$ denotes the relative weight of importance of the score and $f_j$ is the corresponding score function. A candidate $c_i$ is evaluated in terms of score function. The total score *qscore* is defined as follows:

$$qscore_{c_i} = \sum_{j=1}^{n} w_j f_j(c_i)$$

For the reasonable weight, we may empirically configure the value on each strategy.

## 5 PRACTICAL CHALLENGES

In this section, we discuss how we address some important practical challenges applying to real cases.

**Comparing Apples and Oranges.** Since we compare only the control programs, except those, all other components and running environment should be identical. That is, for the reasonable comparison, we need to perform differential testing of two different programs under the same condition. In our experiments, we set up two virtual machines with the same hardware configuration and run those in the same machine. For the same simulated hardware, we use a popular UAV model, 3DR IRIS+, using the same URDF. In Gazebo simulator, URDF describes the UAV model such as the kinematics and dynamic properties of the model, attached sensors including IMU, GPS, etc. We select the common flight modes supported in the both control programs and generate test scenarios which perform the same high-level behavior and then use the identical communication protocol and input in both simulation setups. Then, both programs are run in parallel. CPSDiff collects test results after completion of the input missions.

We use the UAV case as our subjected target in this paper. However, CPSDiff is able to test any type of CPS system, since Gazebo supports any kinds of robotic simulation with URDF. Gazebo provides different types of hardware as plugins and a user can also define custom models.

**Handling Large Input Space.** Our input space is large and continuous. Searching entire space is computationally infeasible with limited computing resources. Since our goal is to compare two control programs in the given scenarios, we do not use the entire space. Especially, in UAV case, flight area is almost unlimited, and position parameters, one of the input parameters, should be limited. To address this issue, we set a *virtual fence* to limit the flight area.

Even though we limit the flight area with *virtual fence*, input space is continuous and large. To search the maximum difference in such a large space, we consider a way of controlling the degree of exploration versus exploitation. Our algorithm is explorative than exploitative. We run multiple iterations with new inputs and select more difference-revealing inputs with a heuristic 'test strategy'. In §6, our experiment results will show that our algorithm outperforms the other two algorithms: random (extremely explorative algorithm) and hill climbing (exploitative algorithm).

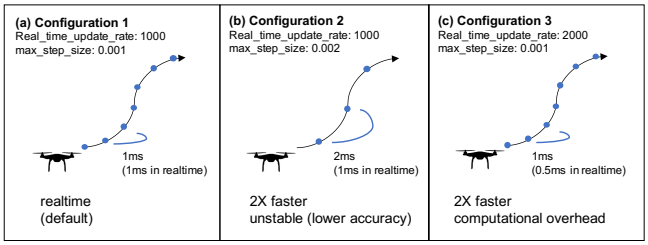



**Figure 10: Simulation time configurations. different update_rate and step_size setting: (a) real-time, (b) 2x-faster with the double step size, and (c) 2x-faster with the double update rate.**

**Simulation Acceleration.** CPS Simulation is computationally expensive and time-consuming. Our algorithm runs a simulation per each iteration, which takes approximately two minutes. For the larger number of iterations, for example, 1,000 iterations, the total execution will take over one and a half days.

Obviously, the larger number of iterations, the more likely to find the best output in the search algorithm. In order to increase the number of iterations, we have to spend a longer total testing time or given the same total testing time, we can decrease the time taken by each iteration. We take two approaches to reduce the simulation time. One option is to speed up a simulation itself by using a simulation timer instead of real-time simulation. Gazebo supports the simulation clock in the world configuration with two parameters: `real_time_update_rate` and `max_step_size`. As shown in Figure 10, `real_time_update_rate` determines the rate at which physics updates are taken per second. Along with `max_step_size` parameter, the time duration in seconds of each physics update step is determined. The product of two parameters represents the target real-time factor, or the ratio of simulation time to real-time. For example, as in configuration 3 in Figure 10, if we set `real_time_update_rate` to 2000 (update every 0.002 second), with a `max_step_size` of 0.001 (one step is 1ms), then our simulation will run 2x faster than real-time simulation. The second option is adjusting the system timer. We use virtual machines to execute two simulations in parallel. By default, the virtual machine keeps all time sources synchronized to the time of a host system. With the tuning timers and time synchronization mechanism in the virtual machine, the guest clock can be accelerated. VirtualBox [41] has a built in feature for this to accelerate the guest clock with a VBoxManage command-line interface and the parameter value of the rate of the virtual clock.

We need to note that the above time-related configuration should be carefully chosen with preliminary experiments. Otherwise, the simulation will generate the results with poor accuracy or severe

computational overhead and thus give unstable time values. The configuration 2 in Figure 10 speeds up the simulation 2 times faster, but the increased step size reduces the accuracy of simulation because the coarse-grained update looses intermediate states between the updates, while the configuration 3 gives limited speed-up because of heavy computation in limited hardware performance. `real_time_update_rate` would cause more frequent update on the limited machine. Because of that, the maximum allowable acceleration is also limited. With this configuration, we accelerate the simulation maximum 2 times faster than the real-time simulator with stable simulation. This helps to reduce the total simulation time and thus increase the number of iterations to search more inputs in the given time.

## 6 EVALUATION

In this section, we present the experimental evaluation of our search algorithm in CPSDiff.

### 6.1 Implementation

**Objective Function.** For the evaluation purpose, we have implemented a prototype of CPSDiff using Python. The prototype supports a differential testing of the battery consumption by two UAV control systems under test: ArduCopter and PX4. The testing involves a computation of a difference value for each test input. In the tool, this difference value is defined as a relative value as given below:

$$\text{Diff} = \frac{\text{abs}(\text{ConBatt}_{\text{Arducopter}} - \text{ConBatt}_{\text{Px4}})}{\text{min}(\text{ConBatt}_{\text{Arducopter}}, \text{ConBatt}_{\text{Px4}})}$$

The amounts of consumed battery, $ConBatt_{Arducopter}$ and $ConBatt_{Px4}$ are calculated by our battery model described in §4.3. We implement the model as a plugin of the Gazebo simulator. The difference value, *Diff*, is computed by the objective function which obtains the amounts of consumed battery from the log files generated by the ArduCopter simulation and the PX4 simulation respectively.

**Algorithms.** The prototype includes an implementation of our *Adaptive Random* (AR) algorithm that is presented in §4.4 with our proposed test strategy. Further, in order to compare the performance of AR, we implemented two more search methods:

- Hill-Climb with Random Restart (HCRR): This is a popular meta-heuristic approach [27], where a hill-climb is an iterative method, which attempts to find an optimal solution by searching in the nearby region of a current good solution. In each iteration, the hill-climb selects the next best input from the current best input and a neighboring candidate input. In our implementation, the neighbor is chosen randomly by tweaking the parameters of the mission commands of the current best input. For that, we add the values taken from the uniform distribution $(-n_p, n_p)$ to the values of the parameters, where $n_p$ is a small percentage (e.g. 5%) of the input range of a parameter $p$. Since, the hill-climb exploits only the neighboring space of the best input, the method performs few random restarts. Each restart initiates a hill-climb from a new randomly selected input and thus allowing to explore the input space.
- Random: This is a purely random input generation method, in which a selection of the next input does not depend on the previous inputs. Thus, this method results into a random exploration of the input space.

**Test Driver.** After an input is generated in each iteration of the algorithm, the test driver of the tool triggers two simulations, ArduCopter and PX4, with that generated input. While the simulations are running, CPSDiff collects flight information from the simulation logs. At the end of both simulations, it computes the *Diff* value as described above and then this value is used by the algorithm to decide about the next input. During the experiments, we observed that sometimes a simulation gets stuck for some reason which we could not handle. Since the tool execution is dependent on the completion of the simulations, we set the timeout after which a stuck simulation is aborted and the corresponding iteration is excluded from the result computation. The timeout value is configured based on the maximum time required by a correct simulation in the experiment.

### 6.2 Experiment Setup

We performed experiments on each of the three algorithms: 1) AR, 2) HCRR, and 3) Random. We run both ArduCopter and PX4 simulation on Ubuntu 64-bit virtual machines with Intel(R) Xeon(R) CPU E5620 @ 2.40GHz x8 processor and 3.8 GB RAM.


**Figure 11: Virtual fence for our experiment**

**Input Space.** We tested the *autopilot* flight mode of our systems under test (SUT). To generate a mission for the *autopilot*, we used four commands that are supported by both of the SUT: *takeoff* (`T`), *land* (`L`), *waypoint* (`WP`) and *loiter for time* (`LO`). Moreover, we used the same input space for all our experiments. First, we fixed the length of a mission command sequence to six, where the first and the last commands in a sequence are always `T` and `L` respectively, whereas a sequence of remaining four commands is a random combination of `WP` and `LO` commands. Second, in order to reduce the simulation time, we have defined a *virtual fence* as shown in Figure 11. The length and width of the virtual fence are both 500 meters, each with the *latitude* parameter of range [40.4220, 40.4224] and the *longitude* parameter of range [−86.9325, −86.9319]. It has the height of 15 meters with the *altitude* parameter of range [5, 20]. Moreover, we have set the range for the *delay* parameter to [0, 5] seconds. The *delay* parameter is used by the `WP` and `LO` commands to specify the time spent by the UAV at the location (latitude, longitude, and altitude) before executing the next command in a sequence. Finally, each mission takes off from the same home location as indicated in Figure 11.

With the above settings for an input space, a software-in-the-loop simulation (ArduCopter as well as PX4) in each iteration of our experiment requires around 2-3 minutes of execution time.

**Number of Input Missions.** We evaluated the performance of the algorithms for 100 input missions. In order to generate missions, AR is configured to iterate 100 times, where in each iteration, thirty heuristically computed waypoints are considered as candidates while generating the next input mission. Random is configured to iterate 100 times, where each iteration generates a completely random next input mission. HCRR is configured to restart 10 times and to perform a hill-climb using 9 iterations after every restart, where a completely random input mission is generated at each restart (=10 missions) and 9 missions are generated from the neighboring input space of the random mission after each restart (=9×10=90 missions).

Further, in order to see if the algorithms show a similar performance when they are run for a longer time, we increased the number of missions to 1,000. For that, AR and Random are configured to iterate 1,000 times, and HCRR is configured to have 1,000 iterations using 20 restarts and a hill-climb with 49 iterations after every restart.

Since the algorithms use some proportion of randomness, we repeated the experiment of 100 missions five times and the experiment of 1000 missions twice for each of the three algorithms. s.

## 6.3 Results and Discussion

Our goal in the experiment is to efficiently find an input i.e., a mission, that would demonstrate as large as the possible difference in the battery consumption of ArduCopter and PX4.

Each algorithm takes around 4 hours on our system to complete an experiment of 100 inputs. At the end of the five trials, on average, AR is able to find the larger difference (46.84%) compared to HCRR (35.01%) and Random (34.37%). Similarly, the experiment of 1,000 inputs, that takes around 42 hours to complete on our system, too shows that AR is more efficient than the other two algorithms: on average, it is able to find the larger difference (61.01%) compared to HCRR (38.06%) and Random (45.67%). In other words, for the given number of iterations, AR seems to find the input mission of our interest (the mission that results in the largest difference among all the missions generated by all the three algorithms) comparatively faster than the other two algorithms.

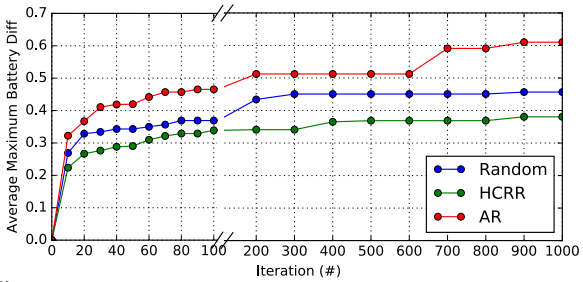
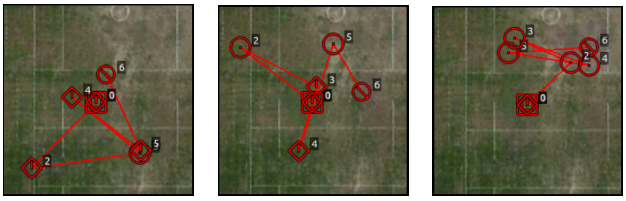



**Figure 12: Average maximum battery consumption difference.**

Figure 12 shows a graph of averages of the maximum battery consumption difference values for each algorithm. The left part of the graph (until $x$=100 iterations) represents the averages computed by combining the results of both sets of experiments: the five trials of 100 iterations and the two trials of 1,000 iterations, whereas the right part (after $x$=100 iterations) represents the averages of the results of 1,000-iterations experiments. The *(x,y)* point marked by a circle in the graph indicates that $y$ is the average maximum difference value until $x$ number of iterations of the algorithm. E.g., for AR, the average of the maximum difference value found until 60 iterations of the five experiments of 100 iterations and two experiments of 1,000 iterations are 44.19%. It can be observed from the graph that AR takes less number of iterations to search a large difference value: e.g. a difference value of 40% is found within only 30 iterations of AR, whereas the number iterations required to reach this difference value by HCRR and Random is greater than 100.

**Best Result Input Mission.** Figure 13 shows the flight trajectories (2D) corresponding to the best input mission (i.e. the mission of maximum difference value) searched by each of the three algorithms. The major visual difference between the trajectory of AR input and the trajectories of HCRR and Random inputs is the sharpness of turns. Every single turn in the AR input trajectory makes the UAV to fly in almost opposite direction, as opposed to those in the HCRR and Random, which do not show very drastic changes in the flight direction.

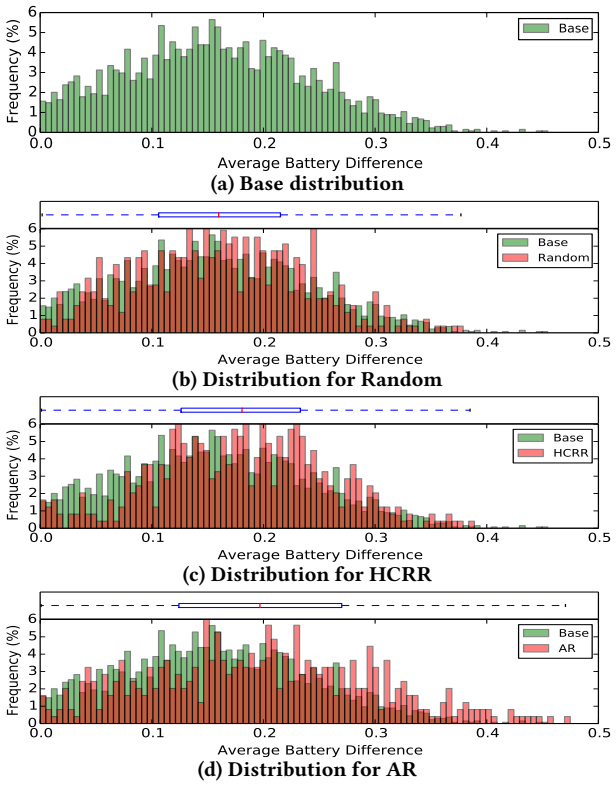


(a)Random (b)HCRR (C)AR

**Figure 13: The best input mission found by each algorithm after 100 iterations.**

**Distribution of Battery Consumption Difference.** We also performed an input space exploration experiment with the intent of studying a possibly entire distribution of the difference values. For that, we executed Random for a large number of iterations: in particular, for the 3,000 purely random input missions. Figure 14a shows the distribution of these 3,000 *Diff* values. It is observed that the difference values are mostly less than 35%, with around 7% of 3,000 difference values greater than 35%. We use this distribution as a base to compare the distributions of the difference values found by the 100-iterations experiment for each algorithm.

**AR vs. HCRR and Random.** According to the above experiment results, we see that AR outperforms HCRR and Random. Distributions of *Diff* values in Figure 14 show the *difference-revealing ability* of each algorithm. It can be observed that AR has a better ability to reveal larger difference values compared to other two algorithms: more number of input missions that lead to *Diff* greater than 35% are found by AR; moreover, as per the spread of the last quartiles of the box-and-whisker plots in Figure 14, AR tends to find more cases of large difference values (>27%) than HCRR and Random. We think that AR is comparatively more effective in finding the



(a) Base distribution
(b) Distribution for Random
(c) Distribution for HCRR
(d) Distribution for AR

**Figure 14: Distribution of battery consumption difference values, *Diff*, found by 100-iterations experiments for each algorithm.**

input cases of our interest because it is able to guide the search of flight trajectories that contain sharp turns using its *Strategy2*. This strategy (described in §4.4) incorporates sudden behavioral changes such as sharp turns in a flight trajectory, which leads to the execution of more control operations in the controllers, and thus are the likely situations for exposing the performance difference in the different implementations of controllers. Our AR implementation takes the sharpness of turns into account while selecting each waypoint in the trajectory from $k$ random candidates. Whereas, in the case of HCRR and Random, we cannot guarantee the presence of sharp turns in the trajectories. In HCRR, each random restart selects a completely random shape which may not contain sharp turns. Moreover, a hill-climb starts with this trajectory of random shape and only finds the neighbors of the waypoints in the trajectory: since the selection of neighbors also is completely random, it may not change the sharpness of turns very quickly. Also, in Random, each trajectory is of completely random shape, which may not contain sharp turns.

## 6.4 Further Research

**Input Minimization.** Our current input uses a sequence of random mission commands. In the experiments, we used a long enough length of a mission command sequence in order to generate all possible combinations and we fixed the number. Therefore, in the randomly generated sequence, some of the sub-sequences may be duplicated or not useful to expose differences because both SUTs show the same behavior under that sub-sequence. We can define the *critical mission* as one that the minimum number of mission sequence which has most difference-revealing ability out of an entire mission sequence. While the entire sequence is still able to expose differences, it takes resources, and identifying the *critical mission* is beneficial. CPSDiff are able to measure intermediate energy consumption at each command completion point. The extension of our tool will support the function to identify the *critical mission*.

**Simulation Speed.** Our current simulation is computationally expensive. Although we can configure physics properties in Gazebo to increase *real-time-update-rate* and accelerate simulation 2-3 times, the simulation time is still a bottleneck for a large number of search iteration. In addition, when we increase *step-size*(resolution of the time window) of the physical update, we encountered unstable simulation with lower accuracy because of coarse-grained state update. In our future work, we plan to provide a solution to address this issues with help from control theory. We believe that the use of an adaptive step size based estimation of the local error and extrapolation can significantly reduce the computational cost. This will be implemented as a plugin of the simulator.

# 7 RELATED WORK

**CPS Testing.** Control software is the core element in CPS, which includes several components such as controllers, network, and many intelligent features for autonomous operations. Especially, the control software has a large number of control parameters which are required to be well configured to give higher control performance. Testing and analysis of the control software in such complex CPS presents a big challenge. Although both cyber and physical domain have many research works separately, little research on testing methods for CPS has been performed. The core of control software, controllers, have been widely studied in theoretical control domain [14, 35]. They mostly focuses on the controller tuning [13, 36] and optimization [19, 33] rather than the automated system testing. Another approach is formal verification of CPS. Model-checking methods [15, 40] have been used in Simulink models. Many model-based testing approaches have been applied to control models in Simulink[28, 30, 37, 42]. These techniques mainly rely on the models to generate test cases and oracles. However, none of these techniques are directly applied to control software and sufficient to demonstrate reliability in real implementation since there are always a gap between models and real implementations. The target of our work is control software which includes not only core controller but also autonomous features (e.g. path planning) in cyber domain. We test the control software within the integrated system through accurate simulation of the system.

**Energy Efficiency Testing.** Energy efficiency is an important issue especially for battery-powered systems because devices cannot operate for a long time without energy consumption optimization. In order to handle this issue, there have been some energy-aware researches. Li et al. [26, 26] focus on storage energy consumption for mobile systems. They tested energy consumption on storage with different software storage stacks based on real power consumption. Then, they figured out which software stack is a major cause of

energy overhead and proposed storage energy models and optimization. During testing, they used typical IO operations (sequential, random read and write) with microbenchmarks and 5 test scenarios with real applications. WearDrive [23] provides energy-efficient storage operation for wearable devices by distributing offline computation to phone. With some typical wearable workloads, they tested energy benefits. AppScope [43] is an energy consumption measurement framework on Android platform. To estimate the energy consumption of individual Android applications, AppScope monitored application's hardware usage at the kernel level with a pre-defined operation sequence. [25] tested energy properties of an Android app with minimized test-suite. Some battery related works [20, 31, 32] in CPS also build mathematical battery models to predict energy consumption or use additional hardware devices to measure real energy consumption, but these works still rely on a few pre-defined test scenarios.

## 8 CONCLUSION

We propose CPSDiff, an automated differential testing for energy-efficient control software, which utilizes adaptive random search to expose larger differences. The approach has been successfully applied to a real case study and generates test cases to compare different control programs. Our evaluation results show that CPSDiff automatically generate test inputs and our search algorithm outperforms than other ordinary search algorithms (Random and HCRR) to reveal larger difference values.

## REFERENCES


[1] *Business Insider - The Pentagonfis most advanced drone keeps falling out of the sky.* http://www.businessinsider.com/the-pentagons-most-advanced-drone-keeps-falling-out-of-the-sky-2016-1.

[2] *International Business Times - Why do US military drones keep crashing? Pentagon silent over mysterious loss of 20 hi-tech weapons.* http://www.ibtimes.co.uk/why-do-us-military-drones-keep-crashing-pentagon-silent-over-mysterious-loss-20-hi-tech-weapons-1539364.

[3] 3D Robotics - Drone & UAV Technology. https://3dr.com/.

[4] Ardupilot Open Source Autopilot. http://ardupilot.org/.

[5] commaai/openpilot: open source driving agent. https://github.com/commaai/openpilot.

[6] Drone - 3DR IRIS. https://3dr.com/support/articles/207358106/iris.

[7] Gazebo. http://gazebosim.org/.

[8] MAVLink Micro Air Vehicle Communication Protocol. http://qgroundcontrol.org/mavlink/start.

[9] Open Source for Drones - PX4 Open Source Autopilot. http://px4.io/.

[10] Self-driving cars now legal in California. http://www.cnn.com/2012/09/25/tech/innovation/self-driving-car-california/index.html.

[11] The UAV - Unmanned Aerial Vehicle. http://www.theuav.com.

[12] Waymo (formerly the Google self-driving car project). https://waymo.com.

[13] A Ali. 2008. A new objective function for controller tuning. In *IFAC Proceedings Volumes (IFAC-PapersOnline)*.

[14] Kiam Heong Ang, G Chong, and Yun Li. 2005. PID control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology* 13, 4 (July 2005), 559–576.

[15] Jire Barnat, Lubo Brim, Jan Beran, Kratochvila, and Italo R Oliveira. 2012. Executing Model Checking Counterexamples in Simulink. In *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE)*. Masaryk University, Brno, Czech Republic, IEEE, 245–248.

[16] E T Barr, M Harman, and P McMinn. 2015. The oracle problem in software testing: A survey. *IEEE transactions on …* (2015).

[17] Boris Beizer. 1995. *Black-box Testing: Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, Inc., New York, NY, USA.

[18] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. 2011. Testing Web Services: a Survey.

[19] Yuan Cao and Jin Ma. 2010. Research on PID parameters optimization of synchronous generator excitation control system. In *2010 5th International Conference on Critical Infrastructure (CRIS)*. North China Electric Power University, Beijing, China, IEEE, 1–5.

[20] Wanli Chang, Alma Pröbstl, Dip Goswami, Majid Zamani, and Samarjit Chakraborty. 2015. Battery- and Aging-Aware Embedded Control Systems for Electric Vehicles. In *2014 IEEE Real-Time Systems Symposium (RTSS)*. TUM CREATE, Singapore City, Singapore, IEEE, 238–248.

[21] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and T H Tse. 2010. Adaptive Random Testing - The ART of test case diversity. *Journal of Systems and Software* 83, 1 (2010), 60–66.

[22] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2001. Progress on the State Explosion Problem in Model Checking. In *Informatics - 10 Years Back. 10 Years Ahead.* Springer-Verlag, London, UK, UK, 176–194. http://dl.acm.org/citation.cfm?id=647348.724453

[23] Jian Huang, Anirudh Badam, Ranveer Chandra, and Edmund B Nightingale. 2015. WearDrive: Fast and Energy-Efficient Storage for Wearables.. In *USENIX Annual Technical Conference*. 613–625.

[24] Koray Inki, Ismail Ari, and Hasan Sozer. 2012. A Survey of Software Testing in the Cloud. In *2012 International Conference on Software Security and Reliability Companion*. IEEE, 18–23.

[25] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2016. *Energy-aware test-suite minimization for Android apps.* ACM.

[26] Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce L Worthington, and Qi Zhang. 2014. On the energy overhead of mobile storage systems.. In *FAST*. 105–118.

[27] Sean Luke. 2009. *Essentials of Metaheuristics.* available at http://cs.gmu.edu/~sean/book/metaheuristics/.

[28] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2016. Automated test suite generation for time-continuous simulink models. *ICSE* 14-22-May-2016 (2016), 595–606.

[29] William M McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* (1998).

[30] Swarup Mohalik, Ambar A Gadkari, Anand Yeolekar, K C Shashidhar, and S Ramesh. 2013. Automatic test case generation from Simulink/Stateflow models using model checking. *Software Testing Verification and Reliability* 24, 2 (Jan. 2013), 155–180.

[31] E N Moraes and L B Becker. 2012. Energy Profile Evaluation of a Cyber-Physical System. *Computing System Engineering (* (2012).

[32] Elisabete NakonecznyMoraes and Leandro Buss Becker. 2012. Remaining Battery Lifetime Determination in Cyber-Physical Systems. *Procedia Computer Science* 10 (Jan. 2012), 215–224.

[33] North China Electric Power University, Beijing, China 2014. *PID parameter optimization based on fuzzy control.* North China Electric Power University, Beijing, China.

[34] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. 2014. Unit testing of energy consumption of software libraries. *SAC* (2014).

[35] K Ogata and Y Yang. 1970. Modern control engineering. (1970).

[36] Olympia Roeva and Tsonyo Slavov. 2014. PID Controller Tuning based on Metaheuristic Algorithms for Bioprocess Control. *Biotechnology and Biotechnological Equipment* 26, 5 (April 2014), 3267–3277.

[37] M Satpathy. 2008. Randomized directed testing (REDIRECT) for Simulink/Stateflow models. In *Proceedings of the 8th ACM International Conference on Embedded Software, EMSOFT'08*. Science Lab, Bangalore, India, 217–226.

[38] O Starov, S Vilkomir, and A Gorbenko. 2015. Testing-as-a-service for mobile applications: state-of-the-art survey. *Dependability Problems of…* (2015), 55–71.

[39] Inc. The MathWorks. 2017. Simulink - Simulation and Model-Based Design - MATLAB & Simulink. https://nl.mathworks.com/products/simulink.html. (2017).

[40] University of Maryland, College Park, United States 2008. *An instrumentation-based approach to controller model validation.* University of Maryland, College Park, United States.

[41] Oracle VM VirtualBox. 2004-2017 Oracle Corporation. Fine-tuning timers and time synchronization. https://www.virtualbox.org/manual/ch09.html. (2004-2017 Oracle Corporation).

[42] Andreas Windisch. 2009. Search-based testing of complex simulink models containing stateflow diagrams. In *2009 31st International Conference on Software Engineering - Companion Volume*. Technische Universitat Berlin, Berlin, Germany, IEEE, 395–398.

[43] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. 2012. AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring.. In *USENIX Annual Technical Conference*, Vol. 12. 1–14.